NOSQL Design for Analytical Workloads: Variability Matters

Victor Herrero, Alberto Abelló, Oscar Romero

Universitat Politècnica de Catalunya - BarcelonaTech
{vherrero, aabello, oromero}@essi.upc.edu

Abstract. Big Data has recently gained popularity and has strongly questioned relational databases as universal storage systems, especially in the presence of analytical workloads. As result, co-relational alternatives, commonly known as NOSQL (Not Only SQL) databases, are extensively used for Big Data. As the primary focus of NOSQL is on performance, NOSQL databases are directly designed at the physical level, and consequently the resulting schema is tailored to the dataset and access patterns of the problem in hand. However, we believe that NOSQL design can also benefit from traditional design approaches. In this paper we present a method to design databases for analytical workloads. Starting from the conceptual model and adopting the classical 3-phase design used for relational databases, we propose a novel design method considering the new features brought by NOSQL and encompassing relational and co-relational design altogether.

Keywords: NOSQL, DW, big data, relational, co-relational, database design

1 Introduction

Deriving valuable information from raw data is nowadays a priority for most companies [21], which see in today's business the need to effectively monitor and analyse own and external data to predict future trends and make informed decisions. The success of Business Intelligence (BI) and the data-driven society paradigm [15] gave rise to data-oriented companies and the consequent data deluge, which requires non-traditional sources (e.g., logs, sensors, free-text data, images, etc.) to be included in current analytical processes. Such paradigm shift is known as Big Data (BD), a wide concept commonly defined by the so-called 3 V's [11], which enable data analysis in the presence of very large volumes (Volume) of heterogeneous (Variety) data in near real time environments (Velocity).

The data warehouse (DW) is the current de-facto implementation standard in BI, where data is multidimensionally modeled (with a star-join schema), stored in Relational Database Management System (RDBMS), and exploited by two means [10]: OLAP and Data Mining/Machine Learning (DM/ML). Thus, the DW is actually modeled according to OLAP needs, while for DM/ML database dumps are generated from the DW and loaded into specialised tools (e.g., SAS

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-46397-1_4

or R). The two exploitation means are still present in BD as Small (OLAP) and Big (DM/ML) Analytics [19]. However, Small and Big Analytics require a specific management when combined with any of the aforementioned 3 V's. As result, Not Only SQL (NOSQL) databases [22] raised as an alternative.

NOSQL systems focus almost exclusively on performance and are based on distributed database principles, also using flexible data models to reduce the impedance mismatch [2]. Aiming at exploiting the data locality principle [16], they discourage dumping data to a file for DM/ML. Consequently, the data scientist role emerged: a data analyst with strong Computer Science skills able to access NOSQL systems and perform advanced analysis inside them. In parallel, several BD tools were developed for them to conduct both Small (e.g., Hive[1]) and Big Analytics (e.g., SparkR[2]) in BD ecosystems. As result, NOSQL systems must **consider** the access patterns of DM/ML to better design the database. However, there are no systematic design methods for them, and traditional ones cannot be reused as-they-are since they do not consider Big Analytics.

In this paper, we present a novel database design method to support analytical workloads (i.e., Small **and** Big Analytics). Currently, some approaches have discussed how to model OLAP in BD (e.g., [17]) but, to our knowledge, there is no systematic unified modeling strategy also considering DM/ML. To accommodate NOSQL novel features, we build on the ideas in [14], where the relationship between relational and NOSQL databases is shown to actually be different faces of the same coin. They claim for the co-existence of the relational (whose core data structure is the relation) and the co-relational (whose core is a finer-grained structure; such as the key-value) data models. In our approach, we benefit from the well-researched relational design techniques and extend them for the co-relational model, and from DW concepts that also apply in BD.

Currently, NOSQL design is, at best, performed at the logical level, in a performance-wise manner and not following the classical ANSI/SPARC architecture. However, this results to be problematic as BD requirements are more dynamic than in DW. Data scientists frequently ask for new attributes, entities or relationships that were not considered in their statistical models before. Thus, since the schema will change and we cannot assume a complete view of the user needs will be available at design time, we need to accommodate *variability* [9] during the design phase (not only that coming from schema evolution, but also caused by heterogeneity of data). For this reason, we claim for starting the design at the conceptual level and for identifying relevant *subject* areas of analysis [10]. Oppositely, purely performance-oriented solutions work at the attribute level and denormalise relational tables (to avoid joins) and cluster attributes according to their *affinity* [16] (i.e., how often they are queried together). However, such solutions do not accommodate evolution as entity correspondences are not preserved. Therefore, we advocate for a high level subject-oriented (i.e., coarser) design preserving the main focus of analysis, which is further refined according to the characteristics of each subject identified (to accommodate variability inside). Such refinement includes the decision per subject of using either the re-

[1] https://hive.apache.org

[2] https://spark.apache.org

lational or co-relational data model (i.e., the degree of denormalization). Finally, performance is considered at the physical level and, according to the expected workload, each subject is vertically fragmented to improve the effective read ratio [16]. We apply our method to an anonimised real-world BD case study and discuss the pros and cons compared to a purely performance-oriented solution.

Contributions. In particular, our main contributions are as follows:

- We follow the traditional 3-phase design: conceptual, logical and physical.
- Integrate both relational and co-relational design into a single quantitative method, also considering classical DW subject orientation.
- We showcase the use of our method in a real use case.

Outline. Section 2 briefly introduces co-relational data models considered. Section 3 introduces the use case. Section 4 details the proposed method. Sections 5 the impact of our method and Section 6 the related work. Finally, Section 7 concludes the paper.

2 Co-relational Models

The three co-relational models considered in this paper are *key-value*, *document* and *column-family*. Fig. 1 classifies their underlying structures with respect to the schema nature. Schemas are explicit if they are declared, which allows the database to automatically parse the instance data. Explicit schemas can in turn be either fix or variable. In the former, all instances' data follow the same schema, which is globally declared once, while in the latter, instance data is individually embedded with its schema. A DBMS with implicit schema does not manage any information about the instance structure, which is a black-box for the system, and data must be parsed at the application level.

To exemplify those three models, we will use a toy example. Let us suppose the following information: "the city of Barcelona (BCN) has a population of 2,000,000 inhabitants and it is located in Catalonia (CAT)". This could be captured in a single relation (*city*), with three attributes: *name*, *population* and *region*. Using SQL notation, it would look like: `city(name, population, region) VALUES ('BCN', '2,000,000', 'CAT')`.

Key-value stores have the simplest layout. Instance data is stored in tables and represented with a key (i.e., identifier), and a value (i.e., associated data). Neither the key nor the value are tied to a specific format. The former is typically a string and the value a binary object. Thus, no declarative query language and optimizer can be provided to accesss the data, and only simple actions (such as get and put by key) are provided via low-level APIs. Consequently, the application layer is responsible for interpreting each instance, and we consider the schema to be implicit. Our example could be represented as: `['BCN', '2,000,000;CAT']`.

Document stores keep documents within collections (i.e., namespace). A document is a key-value structure where the value is a semi-structured document (typically JSON or XML), that can be seen as a set of entries in the form of (potentially nested) key-value pairs. Thus, the schema is explicit but variable,

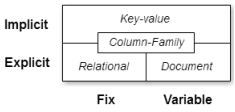

Fig. 1. DBMS based on their schema properties

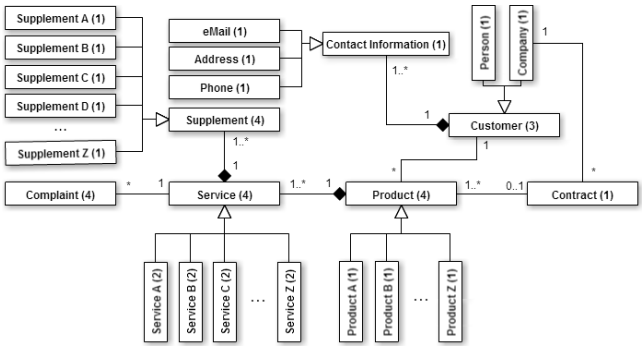

Fig. 2. Use case conceptual schema

since the XML or JSON structure is stored with the instance. In our example, the corresponding document would be `[id:'BCN', population:'2,000,000', region:'CAT']`. So structuring the value opens the door for higher-level query languages and optimization. Typically, document stores use a *query-by-example* approach. Given a pattern document (e.g., *name = 'BCN' or salary > 5000*), all documents fulfilling such pattern are retrieved.

Column-family stores are key-value stores that further structure the value into families, which contain groups of attributes (aka columns). Similar to documents, we could query and retrieve the whole instance, a family, or a specific attribute within a family. Families actually denote vertical fragments, and each is physically stored in a different disk file. From the schema point of view, families are static and defined at table creation time, whereas attributes can be dynamically specified at data insertion time (i.e., the attribute name is stored together with the instance data). Thus, the table schema (i.e., the families) is (i) explicitly declared, static, and shared by all instances, but also (ii) explicit and variable within families, since attributes may vary among instances, and finally (iii) data types are implicit since the attribute value is stored in the form of key-values. Our example could result in a table with two families, namely *population* and *region*, and a constant attribute in each of them, named *value*, to store each attribute: `['BCN', population:{value:'2,000,000'}, region:{value:'CAT'}]`. Alternatively, we could use a family *all* containing both attributes: `['BCN', all:{population:'2,000,000',region:'CAT'}]`. Similar to key-value stores, we may use a single column inside the family: `['BCN', all:{value:'2,000,000;CAT'}]`.

3 Motivating Use Case

This section presents a BD real-world use case where our method was applied to create a decisional NOSQL repository as a complement to the existing relational DW. We choose this use case for being representative of all the problems typically due to variability in BD projects. We outline the limitations of traditional approaches, and define the objectives to be tackled by a design method.

Fig. 2 shows the conceptual schema of the use case domain (due to a disclosure agreement, class names have been altered but relationships between them remain the same). `Customers` (either individual `Persons` or `Companies`) buy `Products` that are composed of `Services` which, in turn, are complemented by different `Supplements`. `Complaints` can be filed if services do not fulfill the customer expectations. To contact customers, their `Contact Information` is registered in the form of `eMails`, `Addresses` or `Phone` numbers. Finally, companies can agree on `Contracts` that comprise several products. Our company needed to **predict** actions from customers regarding products.

The ***large heterogeinity*** involving some entities becomes the first limitation (e.g., dozens of products, several hundreds of services and some thousands of supplements). Relational tables provide a homogeneous representation of entities and we would need to either create a table for each possible *specialised* entity or a single *general* table containing the union of all attributes. This would poorly perform since deploying thousands of tables, that would be joined to produce the general entity, or creating a table with thousands of attributes (`Supplement` alone has almost one hundred) results unpractical and generates expensive queries [22]. Furthermore, new products are constantly developed and released. In the OLTP system, this was solved by implementing specialisations with ad-hoc tables containing generic columns storing different attributes depending on the specialised entity. For example, the `product` table contained hundreds of columns of type `varchar(50)`. A row representing `productA` stores in column `C` the `product model`, while those of type `productB` use `C` to store `location`). A dictionary at the application level keeps track of the mapping of each `product` column (e.g., `C`) to its real meaning (e.g., `productA` $\rightarrow$ `model`).

Schema evolution becomes extremely important in the context of Big Analytics as analysts constantly look for new patterns and therefore ask for new data to be included in the decisional datasets. Reflecting such changes in the relational model is possible, but turns out to be costly as it either requires to alter the current table (massively updating the new columns for existing instances) or create a sibling one (same key) for the new attributes (this was the approach followed in the data sources of our company). Thus, in our use case, changes affecting the DW were simply ignored, resulting in data scientists spending most of their time collecting data from different sources, and cleaning and merging them by themselves prior to conduct the analysis.

Data matrices are the query output for Big Analytics. Traditional DW relies on star-join schemas [12] and data is organised in factual and dimensional tables, which represent subjects of analysis (e.g., `Sales`) and analysis facets (e.g., `Store`, `Time`), respectively. Dimensional data contains hierarchies representing the different levels to which aggregate factual data for each facet. This, e.g., allows aggregating to count `sales` per `store` and `year`, and later easily disaggregate them per `day`. However, star-join schemas cannot be easily used to produce data matrices, because joins between different cuboids are necessary. Consider `Product` to be the subject and `Customer` one of its facets, and we want to produce a flatten data matrix where each row represents a product bought

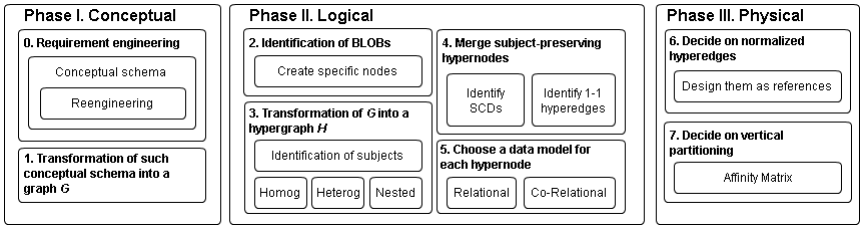

Fig. 3. Summary of steps composing the design method

by a customer. The matrix columns hold any type of information regarding such event, and limitations arise when they must contain data at coarser levels, like `totalAmountBought` (i.e., the total amount bought by the customer of the product in a given matrix row). Since this is not an atomic value, but an aggregated one through the `Product-Customer` relationship, a parallel aggregation on `Product` per `Customer` must be computed and subsequently joined so that all products from the same customer show the same value. Such requirement is usual in DM/ML, but goes against the star-join query pattern [12]. In BI, this issue is tackled once data is loaded in specialised software (such as R) and never considered when designing the database.

Design Objectives: Given those limitations, we present the list of objectives to be overcome by considering co-relational data models.

(a) Simplify the representation of large specialisations so that queries on such entities are kept simple.
(b) Consider schema evolution at design time and acknowledge the possibility of adding new features or values later.
(c) Generate flatten data matrices (i.e., without nested structures).
(d) Performance must be considered first-class citizen.

Note that (a) and (b) correspond to dealing with variability.

4 Design Method for Relational and Co-relational

In current BD settings, the lack of know-how to address database design results in most solutions being designed only considering performance. Oppositely, our method advocates for a top-down approach: we drive the design from the conceptual schema and find a physical design resilient to variability while performance penalisation is minimised. Spanning the 3-phases shown in Fig. 3, in the first phase we assume that a requirement engineering (RE) process, tailored for analysis-oriented systems [7], has been conducted. During RE the conceptual schema must be produced (in BI/BD settings typically by using reengineering techniques [3]) and entity evolution likelihood quantified. Such *quantification* is typical of DW, where it is used to identify Slowly Changing Dimensions (SCD) [12]. Note we assume a correct RE process was conducted and thus, although the conceptual schema may evolve, **the current knowledge is correct**. Starting from the conceptual schema, the second phase decides the degree of normalization after identifying subjects of analysis and, accordingly, proposes a relational

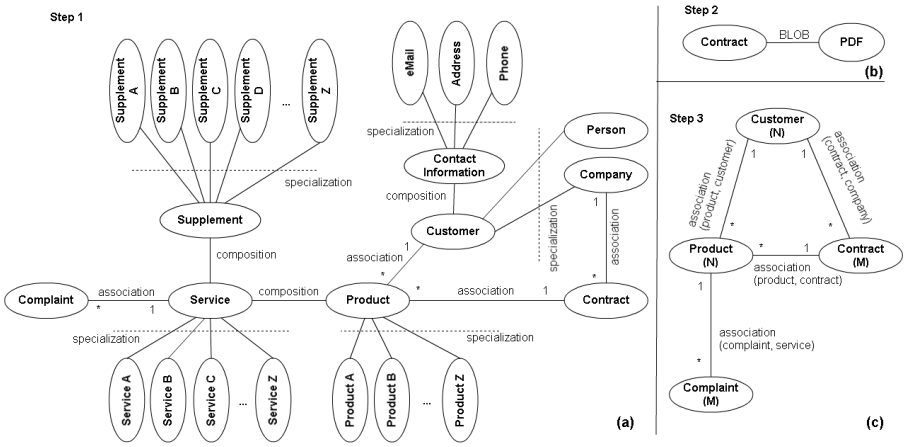

Fig. 4. Graph representation after the first three steps

or co-relational data model. The last phase accommodates performance issues and, according to the currently known workload, vertically fragments the identified subjects to improve the effective read ratio [16].

4.1 Phase I: Conceptual Schema

Step 1. Firstly, we transform the conceptual schema (assumed to have been reified) into an undirected graph $G = (V, E)$ where nodes V denote entities, and edges E denote relationships between them (tagged with the relationship **type** and its **multiplicity**). We consider three relationship types (i.e., "*specialisation*", "*composition*", or "*association*"). Aggregations, unlike compositions, cannot guarantee membership between entities and for our goal they are treated as "associations". Finally, multiplicities are also kept in G.

Use case. We produced the conceptual schema by reverse engineering from the available data sources. The result is depicted in Fig. 2 (numbers by the entities denote their evolving likelihood, being 4 the highest probability), and its transformation into G is shown in Fig. 4(a).

4.2 Phase II: Logical Schema

Step 2. Binary large objects (BLOBs), such as images, videos or other non-detachable objects, cannot be decomposed in smaller subcomponents and are directly understood by the application (i.e., its schema is implicit). Thus, key-value stores rise as a natural option to store them. Any entity containing a BLOB entails the creation of a new node v_{BLOB} in G. This is linked to the entity node v by means of a edge of type "*BLOB*". Although this is a physical decision, since BLOBs are separated because of performance reasons (i.e., unknown format, large size, and rarely retrieved together with other attributes), doing it earlier does not affect the result and simplifies the process.

Use case. BLOBs could be found in `Contract`, where PDF documents were stored. Fig. 4(b) shows how the `Contract` vertex is updated accordingly.

Step 3. We then explore the conceptual schema to identify two different types of entity sets: first, sets of *nested entities* and, second, sets of *heterogeneous* entities. Nested entities essentially refer to compositions where the content can only exist within the container's scope (compositions). Heterogeneous entities are those where the schema may vary among instances (specialisations). Thus, **general entities** (e.g., `Contact Information`) are narrowed in other **specialised entities** (e.g., `Address`, `Phone`, etc.). Entities not involved in any specialisation are considered homogeneous as their schema is fix. Thus, the goal of this step is to group entities regarding the aforementioned types and synthesise G in groups of independent domain concepts (i.e., subjects). This process accordingly results in a hypergraph $H = (X, E')$ where a hypernode $x \in X$ maps to a subgraph of G representing each group entity and E' represents the set of hyperedges. Note an entity can be part of several heterogeneous hypernodes if involved in specialisations belonging to different groups. In these cases, such entity must be replicated in each hypernode. Hypernodes are adorned with its type: Nested, heTerogeneous or hoMogeneous. Also, they take the name of their *main* entity; either the *container* entity from compositions or the most *general* entity from specialisations. Note a hypernode might be adorned with more than one type. We define a dominant function $\gg$ among the tags as follows: $(N) \gg (T) \gg (M)$ so that only one tag prevails over others. Hyperedges E' are created from "associations" in G, from where they inherit their multiplicity.

Use case. Fig. 4(c) shows H. The hypernodes in H (named after the main hypernode entity) contain the following entities:

$X_{customer} = \{$`Customer`, `Person`, `Company`, `Contact Inf`, `eMail`, `Address`, `Phone`$\}$
$X_{product} = \{$`Product`, `Service`, `Supplement`, `(plus all their subclasses)`$\}$
$X_{contract} = \{$`Contract`$\}$
$X_{complaint} = \{$`Complaint`$\}$

Step 4. We now identify hypernodes to be potentially merged in order to improve performance. Note, however, this compromises the variability resilience as modifications in a graph node will impact on the hypernodes it has been placed in. To prevent this, we only merge hypernodes detected as part of the same **subject**. Following well-known DW principles [12], only hypernodes connected by hyperedges with *1-1* and *1-** multiplicities are considered. Merging hypernodes related by *1-1* hyperedges clearly preserves the subject, and the main entity in this case corresponds to the hypernode with higher likelihood to evolve (as it facilitates further changes in the merged hypernode schema). Similarly, hypernodes related by a *1-** hyperedge can only be merged if the to-one end of the hyperedge represents an SCD. Since the SCD evolution likelihood is very low, the main entity of the merged hypernode is its counterpart hypernode in the hyperedge. Replication of an SCD in different merged hypernodes can occur if such SCD is connected to several hypernodes. Finally, the created hypernode acquires the most dominant adornment of the merged ones.

Use case. In the use case, no *1-1* hyperedge exists. `Contract`, however, was identified as an SCD. We thus merged $X_{product}$ and $X_{contract}$ and created

$X_{product\_contract}$. The merged hypernode is adorned as (N) and `Product` is accordingly identified as main entity. Incident hyperedges on the merged hypernodes are now related to $X_{product\_contract}$.

Step 5. For each hypernode, we decide whether it should be designed following the relational or co-relational model depending on the adornments defined. We use the evolution likelihood threshold t_e as indicator to resolve situations where more than one solution is possible.

1. Hypernodes adorned with (M) should be designed by means of relational structures unless they are expected to evolve, as their fix schema can be separately declared and shared by all instances. Oppositely, if their evolution likelihood is above t_e, then they should be designed with co-relational structures to facilitate the accommodation of schema changes.
2. Hypernodes adorned with (T) can be either represented by relational or co-relational structures. Several alternatives exist in case of the former [4], while the most natural way to model heterogeneous entities in case of the latter is with explicit but variable schemas. Deciding between the two data models depends on the degree of heterogeneity. Co-relational should be chosen when the number of heterogeneous entities involved is large (e.g., `Product`). Alternatively, we can consider relational if only few specialised entities exist (e.g., `Contact Information`). In the latter case, two types of schema evolution must be considered: (i) if only new attributes can be added to existing entities, an analogous rationale to M holds, (ii) if new specialisations are regularly created (e.g., new types of `Supplement`), then a co-relational model is a better option. Note that the expertise of the database designer is key to decide, given the RE process artefacts, in which case each hypernode falls.
3. Finally, hypernodes adorned with (N) must be stored in co-relational structures by means of nested lists, which allow recursively storing lists of lists. This way, the container holds its components. In case (N) hypernodes contain entities with fix schema, these are stored in co-relational structures by embedding their schema into the instance. Note that relational structures may also be used, but each of the entities must then be mapped to a relational table and their relationships represented with foreign keys (heavily penalizing performance). Schema evolution in (N) hypernodes is smoothly absorbed by the co-relational model.

   Use case. The data model chosen for the hypernodes are the following:

Co-relational for $X_{customer}$ and $X_{product\_contract}$ (adorned both with (N))
Relational for $X_{complaint}$ (adorned with (M))

4.3 Phase III: Physical Schema

Step 6. We now focus on the remaining hyperedges (i.e., "associations"). Since two data models are being considered, these hyperedges might be relating two hypernodes to be modeled with different data models. Design rules are introduced in Table 1, where all situations are shown. Rows denote what reference

	R-R				*Co-Co*				*R-Co*			
Reference	*1-1*	*1-**	**-1*	**-**	*1-1*	*1-**	**-1*	**-**	*1-1*	*1-**	**-1*	**-**
"->"	✓	✗	✓	✗	✓	✓	✓	✓	✓	✗	✓	✗
"<-"	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓

Table 1. Feasible reference directions for hyperedges

direction is possible given the case in the columns, where R is used to refer to a relational hypernode, and *Co* to a co-relational hypernode. The feasibility of a certain reference is given by considering whether the source hypernode accepts mono-valued or multi-valued attributes, or both. The relational model can only take mono-valued attributes and designing *R-R* hyperedges can be devised in a traditional manner [4], although **-** can also be designed by transforming one end into *Co*. *Co-Co* hyperedges can be indifferently designed, as multi-valued attributes are natively supported. Similarly, for *R-Co* hyperedges, references from the *R*-end are only possible when the *Co*-end has a to-one multiplicity, whereas references from the *Co*-end are always possible.

Use case. (i) The hyperedge relating $X_{complaint}$ and $X_{product\_contract}$ has a **-1* multiplicity relating a relational to a co-relational hypernode. Consequently, this hyperedge is designed through a reference from $X_{complaint}$ to $X_{product\_contract}$. (ii) Two hyperedges relate $X_{product\_contract}$ and $X_{customer}$: one originally relating `Product` and `Customer`, and the other `Contract` and `Company`. Both hyperedges connect two co-relational hypernodes and have a **-1* multiplicity. Thus, they are implemented as two references from $X_{product\_contract}$ to $X_{customer}$ (each reference corresponds to a hyperedge).

Step 7. We aim to improve performance by maximizing the effective read ratio by grouping/splitting entities according to the known workload. Unlike other approaches, we fragment honouring the subjects identified, which cannot be split. Importantly, fragmentation is implemented in most RDBMS and given by all column-family stores. For each hypernode, we identify the vertical fragments checking how often two attributes are queried together. To compute such *affinity*, we can use well-known techniques such as the *affinity matrix* (AM) [16]. Columns and rows in AM represent attributes and a cell describes the frequency these two attributes are queried together. Given a certain threshold t_a, fragments are identified. AM assumes the query workload is given, but this might not be true (e.g., a situation where the solution is built from scratch and there is no past experience on how the database is queried). In such cases, the expected workload is identified during RE. Such process requires the participation of data analysts and techniques such as observation [7].

Use case. For each hypernode, an AM must be created. Fig. 5 illustrates the resulting AM for hypernode $X_{product\_contract}$. Attributes from the use case entities have been renamed to numbered table prefixes in order to honour the disclosure agreement, and cells are percentages. At the bottom of the figure, we sketch a simplified version of the query workload corresponding to four parametrised batch processes creating different matrices. Frequencies represent how many times queries were run per month and we transform them into percentages. *Q1*, *Q3* and *Q4* were executed 20 times/month and *Q2* 15 times/month.

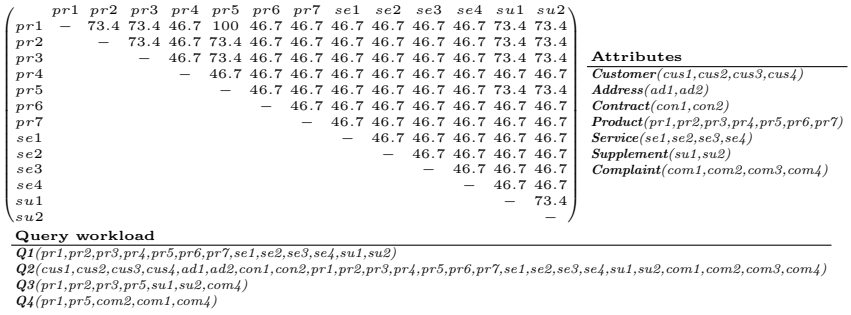

Fig. 5. Affinity matrix for hypernode $X_{product\_contract}$

In our use case, t_a was set to 73.4% and led to two fragments: one containing $\{pr1, pr2, pr3, pr5, su1, su2\}$ and another with the rest of attributes.

Deployment. Importantly, note that deciding relational or co-relational to design a hypernode is not bind to the choice of a specific kind of DBMS, but to unveil its **nature**. Although our method remains agnostic of the chosen product, we finish our case study showing how to deploy the hypergraph in a commercial relational DBMS (i.e., Oracle) and in an open source co-relational store (i.e., HBase plus Hive). The output of our method then hints the best storage model, but the subsequent technological instantiation and the corresponding product-oriented tuning fall out of the scope of this paper.

Despite having a relational architecture underneath, Oracle[3] supports data structures traditionally not considered relational-like. Of special relevance for this paper are the data types *XMLType* and *NESTED TABLE*. The former corresponds to XML data and the latter to tables embedded in other table columns. These data structures map to co-relational structures introduced in Section 2. Documents in Oracle can therefore be stored through the data type *XMLType*, and vertical fragments be implemented with *NESTED TABLE*. Thus, relational hypernodes would be designed as regular relational tables whereas co-relational hypernodes can be designed through *XMLType* structures (if no vertical fragmentation is applied), and *NESTED TABLES* for hypernodes vertically fragmented.

Another example coming from the open-source world is HBase[4] plus Hive[1]. HBase is a column-family system. Consequently, vertically fragmented hypernodes can be naturally stored, regardless being relational or co-relational. Similarly, both relational and co-relational hypernodes where vertical fragmentation did not apply can still be designed as single-family HBase tables. Nevertheless, benefits from using relational structures (HBase has no global schemas and therefore embeds the schema into each instance) and document stores (column values are stored as string and parsing relies on the application level) are then lost in HBase. This cannot be solved from the point of view of the storage, but Hive can be added on top to provide a relational view so that queries can be run as if the underlying storage was relational.

[3] https://www.oracle.com/database

[4] https://hbase.apache.org

5 Scrutinizing Our Method

This section discusses how our method meets design objectives in Section 3.

Objective (a): Our method properly deals with large specializations by means of Steps 3 and 5. In Step 3, entities related by specializations are grouped as part of the same subject. In Step 5, subjects are evaluated to decide the data model to design them. If classified as too heterogeneous, then the co-relational model is chosen. In the use case, for `Product`, `Service` and `Supplement` the number of potential tables was reduced from dozens, hundreds or thousands, respectively, to one entity with explicit and variable schema.

Objective (b): Two key characteristics of our method facilitate schema evolution. Firstly, the main entities from the conceptual model are identified as centroids of a clustered subject-oriented design. Secondly, schema evolution likelihood, quantified per entity in Step 1, is used in Step 5 to decide the data model of each subject. In our use case, we easily added new attributes to entities as well as specialization and composition relationships. During the project we added 205 new attributes/relationships and none required to reconsider the current design.

Objective (c): Conceptually, the multidimensional model is a good starting point for creating matrices. However, the star-join schema statically binds the subjects of analysis with dimensions at design time. To accommodate variability our method identifies subjects (Steps 3 and 5) reflecting them in the database schema. However, unlike a star-join schema, we do not identify dimensions at design time but at query time, depending on analysts concrete needs. Thus, we deploy a *dimensionless* decisional schema, relieving dimensional data of meeting well-formedness OLAP characteristics (e.g., multidimensional normal forms [13]). Decoupling both concepts in the schema provides us with the needed flexibility to tackle unforeseen dimensional concepts. For example, in our use case, several new features were required by data scientists throughout the project. Many times, such features were computed by aggregating data in an already identified dimension but at a coarser granularity, which would have raised the problems discussed in Section 3.

Objective (d): To evaluate performance, we compare the subject-oriented result obtained for our case study (S) against a performance-oriented (P) design for the same workload, built by computing the AM at the attribute level over the universal relation [16]. Considering the same t_a we chose (i.e., 73.4%), we obtain one fragment per entity `Customer`, `Address`, `Contract` and `Service`; plus two more fragments $P_1 : \{pr1, pr5, com4\}$, and $P_2 : \{pr2, pr3, su1, su2\}$, corresponding to a vertical fragmentation of `Product`$\bowtie$`Supplement`$\bowtie$`Complaint`. Despite S only proposed three hypernodes, the number of joins needed is larger than in P, since attribute grouping in P is perfectly tailored to the queries in the current workload (Table 2 reports on the number of joins). Thus, average number of joins of S turns to be 6.3% worse than that of P, a reasonable price for the gain obtained. Note, furthermore, that the effective read ratio of S matches that of P since we apply vertical fragmentation per hypernode (see Step 7).

Query	Frequency	Joins (S)	Joins (P)
Q1	26.7%	0	1
Q2	20.0%	3	4
Q3	26.7%	1	0
Q4	26.7%	1	0
Average:		1.134	1.063

Table 2. Join operations in the subject- (S) and performance-oriented (P) designs

6 Related Work

Operational (write-intensive) RDBMS use normalization to avoid redundancy and therefore insert, update and delete anomalies [8]. Oppositely, decision support (read-intensive) systems use denormalization in order to avoid joins and improve performance. Multidimensional modeling [12], the de-facto standard for DW, is a simple yet powerful metaphor that focuses on subjects of analysis and their facets, which is implemented with a star-join relational schema. However, the star-join schema is not appropriate for flexible BD settings since not only the subject, but also the potential dimensions of analysis are fixed at design time. Furthermore, adding new dimensional or factual data is a costly operation in the DW, since it is typically implemented with relational technology.

Column-oriented engines take vertical fragmentation to the extreme, and redesign the DBMS architecture enabling the combination of light-weight encoding and vector processing [16]. Such engines have shown excellent performance for read-intensive workloads [20] and adaptive systems dynamically exploit vertical or horizontal layouts depending on the workload [1]. However, current techniques for fragmenting a database vertically, such as attribute clustering or AM [16], do not consider evolution and assume static workloads. Also, vertical fragmentation is not always the best modeling choice [1]. Finally, several guidelines specific for NOSQL design are nowadays available [6, 18, 22] presenting high-level guidelines that map either to phase two or three of our method. Other approaches bet for the integration of heterogeneous data by means of functional SQL-like languages [5] and, thus, integration occurs at query time rather than at design time. To our knowledge, this is the first holistic approach encompassing the relational and co-relational design altogether.

7 Conclusions

We have presented a novel method to holistically address the design of relational and co-relational databases in the presence of analytical workloads. Unlike most spread habits among BD practitioners, we underline the importance of the conceptual schema and propose a method resembling traditional database design following the classical 3-phase design: conceptual, logical and physical. However, we do not diminish the importance of performance in BD, but rather balance it with other equally important aspects such as data structural variability, which we have shown that can be managed by subject-oriented design (a well-known DW concept). We have exemplified our method with a real case study paradigmatic of the typical modeling complexities found in BD projects, and shown the benefits of our design approach.

Acknowledgments

We would like to thank Antoni Olivé for revising the paper.

References

1. Alagiannis, I., et al.: H2O: A Hands-free Adaptive Store. In: SIGMOD (2014)
2. Ambler, S.: Agile Database Techniques: Effective Strategies for the Agile Software Developer. Wiley& Sons (2003)
3. Blaha, M.: On Reverse Engineering of Vendor Databases. In: WCRE (1998)
4. Blaha, M.: Patterns of Data Modeling. CRC Press, Inc. (2010)
5. Bondiombouy, C., Kolev, B., Levchenko, O., Valduriez, P.: Integrating big data and relational data with a functional sql-like query language. In: Database and Expert Systems Applications - 26th International Conference, DEXA 2015, Valencia, Spain, September 1-4, 2015, Proceedings, Part I. pp. 170–185 (2015), http://dx.doi.org/10.1007/978-3-319-22849-5_13
6. Bugiotti, F., et al.: Database Design for NoSQL Systems. In: ER (2014)
7. Garcia, S., et al.: DSS from an RE Perspective: a Systematic Mapping. J. of Systems & Software 117 (2016)
8. Garcia-Molina, H., et al.: Database systems - The Complete Book. Pearson Education (2009)
9. Gartner: Focus on the 'Three Vs' of Big Data Analytics: Variability, Veracity and Value, https://www.gartner.com/doc/2921417/focus-vs-big-data-analytics
10. Inmon, W.H., et al.: Corporate Information Factory. Wiley& Sons (2001)
11. Jagadish, H.V., et al.: Big data and its technical challenges. Commun. ACM 57(7) (2014)
12. Kimball, R.: The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses. Wiley (1996)
13. Mazón, J., et al.: A Set of QVT Relations to Assure the Correctness of DWs by Using Multidimensional Normal Forms. In: ER (2006)
14. Meijer, E., Bierman, G.M.: A co-relational model of data for large shared data banks. Commun. ACM 54(4) (2011)
15. OCDE: Data-driven Innovation for Growth and Well-being, http://www.oecd.org/sti/inno/data-driven-innovation-interim-synthesis.pdf
16. Özsu, M.T., Valduriez, P.: Principles of Distributed DB Systems. Springer (2011)
17. Romero, O., et al.: Tuning Small Analytics on Big Data: Data Partitioning and Secondary Indexes in the Hadoop Ecosystem. Inf. Syst. 54 (2015)
18. Sadalage, P., Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional (2012)
19. Stonebraker, M.: What Does 'Big Data' Mean?, http://cacm.acm.org/blogs/blog-cacm/155468-what-does-big-data-mean/fulltext
20. Stonebraker, M., et al.: C-Store: A Column-oriented DBMS. In: VLDB (2005)
21. TDWI: TDWI Best Practices Report, Achieving Greater Agility with Business Intelligence, https://tdwi.org/research/2013/01/tdwi-best-practices-report-achieving-greater-agility-with-business-intelligence.aspx
22. Wiese, L.: Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases. DeGruyter (2015)